STATICROUTE: A NOVEL ROUTER FOR THE DYNAMIC PARTIAL RECONFIGURATION OF FPGAS

Brahim Al Farisi, Karel Bruneel, Dirk Stroobandt

Ghent University, ELIS Department
Sint-Pietersnieuwstraat 41, 9000 Gent, Belgium
{Brahim.AlFarisi, Karel.Bruneel, Dirk.Stroobandt}@UGent.be

ABSTRACT

Using Dynamic Partial Reconfiguration (DPR) of FPGAs, several circuits can be time-multiplexed on the same chip region, saving considerable area. However, the long reconfiguration time when switching between circuits remains a large problem with DPR. In this paper we show it is possible to significantly reduce reconfiguration time when the number of circuits is limited. We tackle the problem by reducing the time needed to reconfigure the FPGA's routing. We divide the configuration memory of the FPGA's routing in a static and a dynamic portion. A novel router, called StaticRoute, is presented that is able to route the nets of the different circuits in such a way that the static portion is shared and only the dynamic portion needs to be reconfigured. The static portion of the configuration memory does not need to be rewritten during run-time. In the experiments we show it is possible to reach a $2\times$ speed-up of the reconfiguration process, while the increase in wire length per circuit is limited.

I. INTRODUCTION

Dynamic partial reconfiguration (DPR) of FPGAs allows designers to time-multiplex several circuits on the same chip area, called the reconfigurable region (RR). DPR makes it possible to use smaller and thus cheaper FPGAs, because FPGA resources can be reused between circuits.

The configuration memory of the RR consists of SRAM memory cells that control the content of the look-up tables and the state of the routing switches. To implement a circuit in the RR, a configuration needs to be generated that contains the binary values that need to be written in the RR's memory cells. In conventional DPR systems, a configuration is generated for every circuit by implementing it separately in the RR. Every memory cell of the RR then corresponds to a collection of binary values, one for each circuit. When these binary values are the same, we call this collection a static bit. Otherwise this collection is called a dynamic bit. Memory cells containing a static bit do not need to be rewritten during run-time.

However, in current FPGAs, the reconfiguration granularity is a collection of memory cells called a frame. A whole frame needs to be rewritten, even when only one memory cell of the frame contains a dynamic bit. The problem with conventional DPR systems is that the dynamic bits are scattered over the frames of the configuration memory, making it necessary to reconfigure the complete reconfigurable region [1]. This leads to long reconfiguration times, making DPR less useful for more dynamic applications [2] [3].

In this paper we propose a novel approach to reduce the reconfiguration time when the number of circuits to be implemented in the RR is limited. It is clear that during reconfiguration most time is spent writing the routing's configuration memory. We therefore focus on reducing the time to reconfigure the FPGA's interconnection network.

Our novel technique consists of two steps. In a first step the configuration memory of the RR's routing switches is divided in a static and a dynamic portion. Care needs to be taken that the memory cells of the static portion reside in other frames than those of the dynamic portion. Then, in a second step, the placement of conventional DPR is retained, but the connections of *all* circuits are routed together using our novel router, which we named StaticRoute. StaticRoute routes the connections in such a way that dynamic bits are avoided in the static switches of the RR. The dynamic bits are thus clustered in the dynamic portion of the configuration memory. To the best of our knowledge, we are the first to propose such an approach.

In this paper we introduce the concept of *switch congestion*. A switch is said to be congested when it is in a static portion, but is controlled by a dynamic bit. StaticRoute is based on the PathFinder algorithm [4] and makes use of the negotiated congestion mechanism to resolve both wire and switch congestion. In our experiments we show that a speed up of almost $2\times$ of the reconfiguration process can be obtained, while the increase in wire length is limited.

Our paper starts with a comparison of the conventional DPR tool flow and our newly proposed tool flow using StaticRoute in Section II. StaticRoute is discussed in more detail in Section III. The experiments and results are discussed in Section IV. Finally, we conclude in Section V.

II. DYNAMIC PARTIAL RECONFIGURATION

With dynamic partial reconfiguration (DPR) it is possible to implement different circuits, that are not needed at the same time, on the same FPGA area. This area is generally called the reconfigurable region (RR). Whenever one wants to change the implemented circuit, an amount of time is needed to rewrite the configuration memory. This is called the reconfiguration time. The subsystem that performs the reconfiguration is called the reconfiguration manager and is generally implemented in software. In this section we discuss two tool flows that use DPR: the conventional DPR flow and our novel approach using StaticRoute.

II-A. Conventional DPR flow

The conventional DPR tool flow implements every circuit separately in the reconfigurable region by following the typical steps of the FPGA CAD flow (synthesis, technology mapping, placement and routing), as shown in Figure 1(a). For every circuit a configuration is generated that contains the binary values needed to write the configuration memory of the reconfigurable region. To switch between the different circuits the reconfiguration manager writes the reconfigurable region with the appropriate configuration.

After implementation, there is a configuration available of the RR for every circuit. Every memory cell of the RR then corresponds to a collection of binary values, each from a different circuit. When these binary values are the same, we call this collection a static bit. Otherwise this collection is called a dynamic bit. If a memory cell contains a static bit, it means it has the same value for the different implemented circuits. Static bits do not need to be rewritten during run-time.

However, in current FPGAs, the reconfiguration granularity is a collection of memory cells called a frame. A whole frame needs to be rewritten, even when only one memory cell of the frame contains a dynamic bit. The problem with conventional DPR systems is that the dynamic bits are scattered over the frames of the configuration memory, making it necessary to reconfigure the complete reconfigurable region [1]. This may lead to reconfiguration times that are too long for more dynamic applications [3] [2].

II-B. Novel tool flow using StaticRoute

In our method, the configuration memory of the RR's routing is split into two parts: a static part and a dynamic part. The proposed tool flow is presented in Figure 1(b). Instead of running the tool flows completely separately for the different circuits, the idea is to have a joint routing of the circuits. In this case, the tool flow is run separately until placement, generating a placed design for each circuit. Then the nets of all the circuits are merged into one set of nets. During this merging step, each net is automatically annotated with the name or ID of the circuit it belongs to.

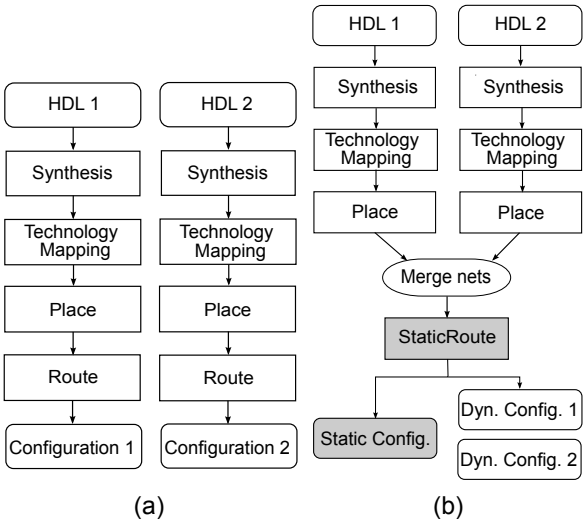

Fig. 1. The conventional DPR tool flow (a), compared to our novel approach which uses StaticRoute (b).

The set of merged connections is then routed with StaticRoute. The information annotated during merging is used by StaticRoute to generate one static configuration and a dynamic configuration for every circuit. The static configuration contains the binary values for the static part of the RR's routing. It only needs to be loaded in the FPGA's configuration memory once at start-up. The dynamic configurations contain the remainder of the configuration of the RR. These are used to switch between circuits during run-time. Since the dynamic configurations are much smaller than a configuration of the complete RR, reconfiguration time can be reduced considerably.

III. STATICROUTE

StaticRoute is based on the PATHFINDER algorithm, the most commonly used algorithm for FPGA routing. In the first part of this section we will therefore first give a brief discussion of PATHFINDER.

Before StaticRoute is used, the routing switches of the reconfigurable region (RR) are split into two parts: a static part and a dynamic part. Then the connections of *all* circuits are routed together in such a way that there are no dynamic bits controlling static switches. To also take the switches into consideration during routing, the representation of the FPGA's architecture needs to be extended. This is explained in section III-B.

Detecting dynamic bits after the configurations are generated is easy. When a memory cell has different values in the different configurations, it contains a dynamic bit. This means it will have to be rewritten during run-time. In section III-C we will, however, show that it is also possible to already detect dynamic bits during routing.

Finally, section III-D handles how StaticRoute extends the cost function of PATHFINDER, so that dynamic bits are avoided in the static portion of the configuration memory.

III-A. The Pathfinder algorithm

A conventional router calculates the Boolean values that need to be stored in the memory cells of the configurable interconnection network so that the physical logic blocks are connected as is specified by the nets in the mapped circuit. The main algorithm used to solve this problem is PATHFINDER [5].

PATHFINDER presents the available routing resources of the FPGA in an easy-to-explore data structure, the routing resource graph (RRG). The RRG is a directed graph, where each node represents a routing wire on the FPGA and each directed edge represents a routing switch on the FPGA[1].

In the PATHFINDER algorithm, the connections that need to be routed are organized in nets. These are sets of connections that share the same source. During the first routing iteration,nets can share resources at no extra cost and thus, each net is routed with a minimum number of wires. In subsequent routing iterations, the algorithm rips up and reroutes all the nets in the input circuit. A wire is said to be congested if it is used by more than one net. The routing iterations are repeated until no shared resources exist or, in other words, the wire congestion is resolved. This is achieved by gradually increasing the cost of sharing resources between nets, a technique called *negotiated congestion*. The cost function of a wire in the RRG is

$$cost(n) = b(n) \cdot p(n) \cdot h(n), \tag{1}$$

where $b(n)$ is the base wire cost (equal to 1), $p(n)$ is the present wire congestion penalty and $h(n)$ is the historical wire congestion penalty. The factor $p(n)$ is updated every time a net is rerouted and is used to avoid wire congestion during one routing iteration. The factor $h(n)$, on the other hand, is only updated every routing iteration. It is used to make heavily used resources in past routing iterations more expensive. In this way a wire congestion map is built, which enables nets to avoid routing through heavily congested wires, if possible. More details on PATHFINDER can be found in [4].

III-B. Extended routing resource graph

In a standard RRG the nodes represent wires and the directed edges represent the switches. StaticRoute does not make use of a standard RRG, but of an extended RRG. This is an RRG where also the switches are represented as nodes. An example of an extended RRG is shown in Figure 2. The round nodes are wires and the square nodes switches.

[1]This is a simplification.The nodes can also represent logical pins or sources or sinks. These are treated the same [4].

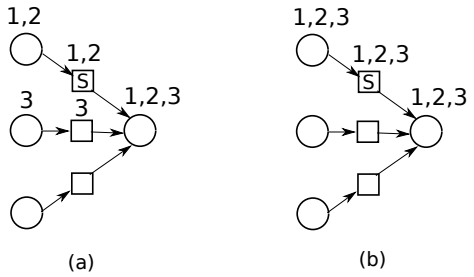

Fig. 2. An example of a switch S controlled by a dynamic bit (a) and one controlled by a static bit (b).

Such a representation is necessary for two reasons. First, it is possible to mark certain switches as being static. The rest of the switches are considered dynamic. This way the RR's routing, and thus also its configuration memory, is split up in a static and a dynamic part.

Second, in an extended RRG certain information can also be associated with the switches during routing. This can be a cost or information on which circuits are using a certain switch.

III-C. Detecting dynamic bits in the extended RRG

As can be seen in Figure 1 (b), the input to StaticRoute is a set containing the nets of *all* circuits that need to be implemented in the reconfigurable region. These nets are all annotated with the ID of the circuit they belong to, as explained in Section II-B. When a net uses a node during routing, it also gets annotated with this information. Our starting point in this section is therefore an extended routing resource graph annotated with the circuits' IDs.

Let us assume 4 circuits, numbered 1 to 4, are implemented in the RR. In Figure 2(a) we see a routing multiplexer of the RR, represented as an extended RRG. It connects the top wire to its output for circuits 1 and 2. The middle wire is connected to the output for circuit 3. Switch S therefore needs to be closed for circuit 1 and 2. It needs to be open for circuit 3, as not to add any extra capacitance of the wires of the other circuits. S has a don't-care value for circuit 4, because this circuit is not using this multiplexer. In this case, S clearly is controlled by a dynamic bit, since it has different values for different circuits.

Let us look at a second example in Figure 2(b). In this case the switch S has value 1 for circuits 1, 2 and 3. And it has a don't-care value for circuit 4. It is clear that when a switch and its connected wires are used by the same circuits, it does not have to be changed during run-time. The switch is closed for the circuits that use it and has a don't-care value set to 1 for the other circuits.

In general, in the extended RRG, a switch node S connects two wire nodes W_{in} and W_{out}. Let us assume that S is used by a set of circuits C_S. W_{in} and W_{out}

are used by C_{in} and C_{out} respectively. We state that S is controlled by a dynamic bit if:

$$((C_S \neq C_{in}) \vee (C_S \neq C_{out})) \wedge C_S \neq \phi. \quad (2)$$

The condition $C_S \neq \phi$ is necessary to exclude unused switches, which are always static.

III-D. Novel cost function

In the previous sections we explained that in the extended RRG some switches are marked as being static. We also presented a way to detect dynamic bits in the extended routing resource graph. In this section we introduce the term *switch congestion*. A switch is said to be congested when it is marked as static, but is controlled by a dynamic bit.

In the PATHFINDER algorithm the cost of using a wire only takes into account wire congestion. The nets are ripped up and rerouted until there are no wires that are congested. In this section we describe how we extended this algorithm to also take switch congestion into consideration. StaticRoute rips up and reroutes the nets of all circuits until all wire *and* switch congestion is resolved. The ordering of the nets is in such a way that the nets are traversed per circuit.

In the PATHFINDER algorithm a connection of a net is routed by searching the path of wire nodes with lowest cost in the routing resource graph. In our algorithm the same happens in the extended RRG. Except that, an extra cost per wire is added, to take switch congestion into consideration. The cost of a node in the extended RRG is

$$cost(n,c) = \begin{cases} cost_w(n,c) + cost_s(n) & \text{if n is a wire} \\ 0 & \text{if n is a switch} \end{cases} \quad (3)$$

When a wire is used, the congestion of all the static switches that are connected with it are affected. That is why the cost of switch nodes in the path of the RRG is zero and the cost addition is made in the wires. Switch nodes are used to hold information needed to determine the switch congestion penalty $cost_s(n)$.

The term $cost_w(n,c)$ takes wire congestion into consideration and is very similar to Equation 1. Remember that StaticRoute routes the nets of *all* circuits together. However, when a net of the circuit c is routed, only the other nets of c are taken into consideration for the wire congestion. This is because nets of different circuits do not cause wire congestion. They are never present on the FPGA at the same time and therefore can share wires. The equation for $cost_w(n,c)$ is

$$cost_w(n,c) = p(n,c) \cdot h(n,c), \quad (4)$$

where $p(n,c)$ and $h(n,c)$ are the present and history wire congestion penalty for circuit c. These are calculated like in [4].

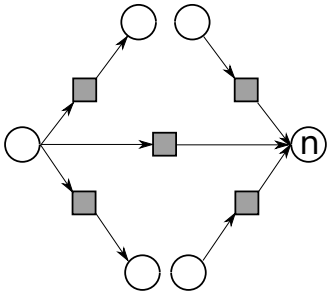

Fig. 3. Example where the switches of the set S(n) for a wire node n are indicated in grey in the extended RRG.

The term $cost_s(n)$ takes switch congestion into consideration. As mentioned in the previous section, to determine whether a switch is congested both the wires it connects are needed. Therefore the router assigns the cost for switch congestion using a set $S(n)$ that is the union of the fan-in switch nodes of the current wire node n and the fan-out switch nodes of the previous wire. In Figure 3 an example is shown where the switches of $S(n)$, associated with a wire node n, are indicated in grey. In this set $S(n)$, we can use Equation 2 to identify a subset of congested switches we will call $C(n)$. The algorithm only takes into consideration the switch congestion caused by the current net. This means the current circuit is in C_{in} or C_{out}.

Given a wire node n, with its associated set of congested switches $C(n)$, we have following equation for the switch congestion penalty

$$cost_s(n) = p_s(n) \cdot h_s(n), \quad (5)$$

where $p_s(n)$ and $h_s(n)$ are the present and history switch congestion penalty. The factor $p_s(n)$ resolves switch congestion during one iteration and is given by:

$$p_s(n) = 1 + |C(n)| \cdot p_{fac}, \quad (6)$$

Note that if the use of a wire results in more congested switches it gets penalized more. The factor $h_s(n)$ takes into consideration the switch congestion that occurred in the previous iterations. It uses the congestion map that is built in the switch nodes. It is given by:

$$h_s(n) = \sum_{m \in C(n)} h_s(m), \quad (7)$$

where $h_s(m)$ is the history switch congestion penalty of one switch node m. This is updated every routing iteration i as follows:

$$h_s^i(m) = \begin{cases} 0 & \text{if } i = 1 \\ h^{(i-1)}(m) & \text{if m is not congested} \\ h^{(i-1)}(m) + h_{fac} & \text{otherwise} \end{cases} . \quad (8)$$

The way the factors p_{fac} and h_{fac} change as the algorithm progresses is called the routing schedule. The same routing schedule is used for both wire and switch congestion [4].

IV. EXPERIMENTS AND RESULTS

IV-A. Benchmarks

To validate our proposed tool flow we conducted experiments using 3 different applications. In the first 2 experiments typical multi-mode applications were used: a regular expression matching (RegExp) and an adaptive filtering application (FIR). In the last 2 experiments general MCNC benchmarks were used.

In [6] a tool was developed that can generate a hardware engine, written in VHDL, that matches a certain regular expression. In the first experiment, we chose 5 middle-sized regular expressions out of the Bleeding Edge rules set [7] and with this tool generated the corresponding circuits. In the second experiment we generated 5 fixed coefficient finite impulse response (FIR) filters. The FIR filters are fully pipelined, have 16 taps and the width of the input and the coefficients is 8 bit. The values for the coefficients were chosen randomly, after which all the constants were propagated. Such FIR filters are 3 times smaller than the generic version.

In the third experiment, we chose 5 circuits out of the general MCNC benchmark suite [8] that were of similar size compared to the rest of the circuits in the previous experiments(MCNC). In the fourth experiment we chose 5 circuits out of the MCNC20 benchmark suite [8]. The names of the MCNC and MCNC20 circuits used in this experiments can be found in Table II.

For every set of circuits the minimum, average and maximum number of LUTs are reported in Table I. In each set all possible 10 combinations of 2 circuits out of 5 were chosen. These combinations of 2 circuits were each time implemented using both the conventional DPR flow and our novel approach using StaticRoute.

Table I. Size of the LUT circuits used in the experiments.

	Minimum	Average	Maximum
RegExp	500	516	543
FIR	235	302	371
MCNC	264	310	404
MCNC20	1135	1323	1544

IV-B. FPGA architecture

StaticRoute was implemented based on our JAVA version of the VPR (Versatile Place and Route) wire-length driven router [4]. VPR is the most commonly used academic tool for place and route algorithms [4] . The FPGA architecture used is described in `4lut_sanitized.arch`. This is an FPGA architecture file included in the distribution of VPR. It has logic blocks containing one 4-LUT and one flip-flop and the wire segments in the interconnection network only span one logic block. Two modifications were made to the routing architecture to better resemble the commercial available FPGAs. Wilton switch blocks and unidirectional wires are used instead of a disjoint switch blocks and bidirectional wires [9].

We note that the techniques and tools we use in this paper are independent of the architecture used. The number of inputs of the LUTs is simply an input parameter of the tool flow. Also, different routing architectures can be used since StaticRoute uses a straightforward extension of a standard representation of the routing infrastructure called the routing resource graph.

Since there is no other functionality implemented on the FPGA, the reconfigurable region comprises the complete FPGA in our experiments. The minimum square area of the FPGA was chosen that fits both circuits.

The average minimum channel width in the industrial benchmark set of VTR is 98 tracks [10]. To determine the order of magnitude of typical commercial FPGAs, Altera's Chip Planner Tool can be used [11]. Using this tool it was determined that routing channels of commercial FPGAs, like the Stratix IV, typically consist of several 100 tracks. The channel width in these experiments was chosen only 50% bigger than the minimum needed. Keeping the relative overprovisioning of channel width constant will allow us to make a fair comparison of the wire lengths of different circuits.

IV-C. Results

We point out that both our tool flow and the conventional DPR flow have the same gains in area. For the regular expression matching application and the MCNC benchmarks, only an area of around 50% is required compared to the static implementation of the 2 circuits. The adaptive filtering application requires an area which turned out to be only 33% of the generic FIR filter.

Two other metrics were used to further evaluate the quality of the implementation: reconfiguration time and

Table II. Name of the MCNC and MCNC20 circuits used in the experiments.

MCNC	e64, rd73, s400, s1238, s1494
MCNC20	apex4, alu4, tseng, ex5p, misex3

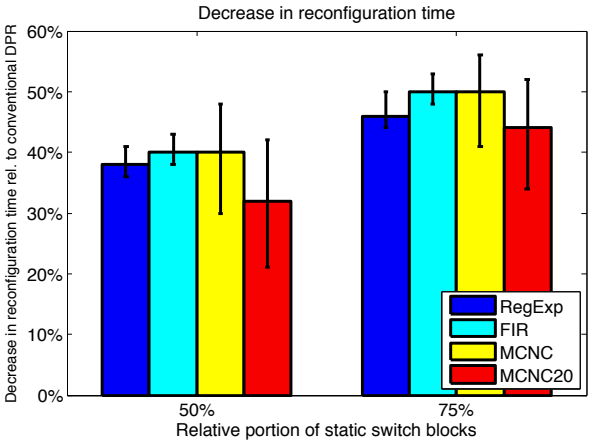

Fig. 4. Decrease in reconfiguration time compared to conventional DPR.

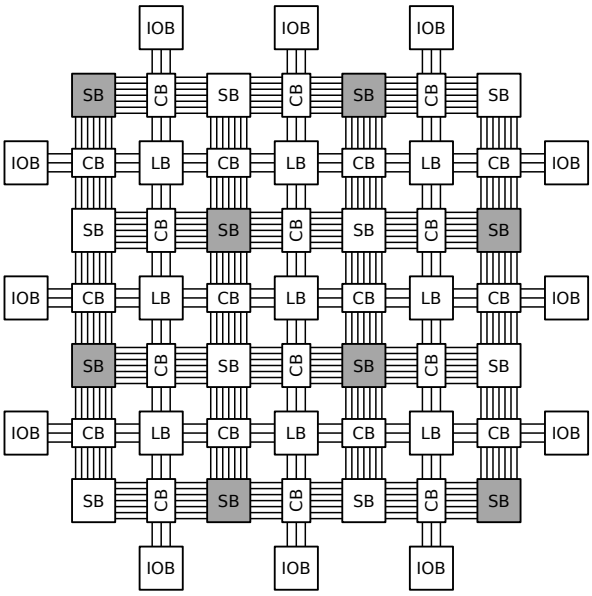

Fig. 5. An example of a 3×3 island style FPGA where 50% of the switch blocks are marked static (in grey).

wire-length. The reconfiguration time gives an indication on how fast the system can adapt when necessary. Wire length is an important metric for the quality of a circuit, since it correlates with power usage and performance (maximum clock frequency) of a circuit [4]. We focus on the effect the relative size of the static portion of the configuration memory has. We average the results over the implemented circuits and use error bars to indicate minimum and maximum values.

IV-C1. Reconfiguration time

Since the experiments were done in our JAVA based version of VPR, there are no configuration frames defined. As can be seen in Figure 5, the wires in an island style FPGA are organized as channels in between the logic blocks (LBs). The switches that connect the wires and the logic block pins are aggregated as connection blocks (CBs) and switch blocks (SBs). The connection blocks connect the logic block pins to the wires in their neighboring channel while the switch blocks connect the wires from one channel to wires from an adjacent channel.

Marking the static bits in the routing infrastructure was done based on the switch blocks. For each set of benchmarks, we compare the cases where 50% and 75% of the switch blocks was marked static. In Figure 5 an example is shown where 50% of the switch blocks is marked static. As can be seen, this is done in such a way that the selected switch blocks are spread uniformly over the FPGAs area. The connection blocks in the routing were all kept dynamic.

In this experiment we look at the *total* reconfiguration time. For conventional DPR this is the sum of the LUT bits and the bits that control the switches. For our novel approach, that uses StaticRoute, we use the same, but don't count the routing bits that reside in the static portion of the configuration memory. As explained earlier, the bits in the static portion are shared by all circuits and thus don't need to be rewritten during run-time.

In Figure 4 the relative decrease in reconfiguration time is shown compared to the conventional DPR flow. Static switch blocks do not need to be reconfigured during run-time. The decrease in reconfiguration time is therefore directly proportional to the relative portion of static switch blocks. It is approximately the same for all benchmarks considered. This is, on average, 40 % when half of the switch blocks are static to 50% for 75% static switch blocks.

IV-C2. Wire length

In our proposed tool flow the different circuits are not implemented separately, as is the case in the conventional DPR flow. Instead, the circuits are routed together using StaticRoute. In this section we assess the impact this has on the wire length. Each circuit uses a set of wires when it is active. We compare the size of this set in the case of implementation with the conventional DPR flow and StaticRoute. This is then averaged over all circuits.

The results are shown in Figure 6. Again, the relative increase in wire length is dependent upon the relative size of the static portion. When 50 % of the switch blocks are marked static, then the wire length increases on average a few percent and the maxima are around 5 %. For some benchmarks the wire length even decreases a little when using StaticRoute, this is because both PATHFINDER and StaticRoute are heuristics. For 75 % static switch blocks the average increases somewhat, especially for the regular expression applications. Also the maxima increase

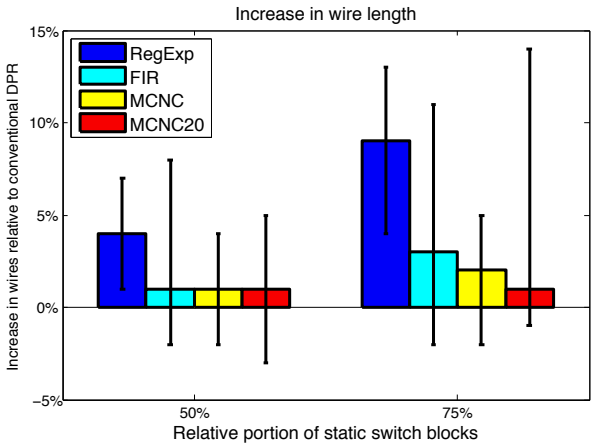

Fig. 6. Wire length increase of StaticRoute compared to conventional DPR.

to around 10 %. We can conclude that the wire length increase of using this technique, when implementing 2 circuits, is limited.

V. CONCLUSION

In this paper we introduced the notion of switch congestion. This occurs when a dynamic bit resides in the static portion of the configuration memory. We showed that it is possible to already detect dynamic bits during routing in the extended routing resource graph. An extended version of the PATHFINDER algorithm, called StaticRoute, was presented that was used to route all circuits together. It is able to resolve both wire and switch congestion. Therefore, using StaticRoute, the dynamic bits are no longer scattered over the complete configuration memory of the routing. Instead they are clustered in the dynamic portion. To the best of our knowledge we are the first to propose such a method, which can be used in a frame based reconfiguration approach. In our experiments we showed that, using StaticRoute, a 2× speed up of the reconfiguration process can be obtained, while the increase in wire length is limited. In the future we would like to implement our novel tool flow on a commercial FPGA. We are also looking at ways to automatically mark the static portion of the routing's configuration memory.

VI. REFERENCES

[1] P. Sedcole, B. Blodget, T. Becker, J. Anderson, and P. Lysaght, "Modular dynamic reconfiguration in Virtex FPGAs," *Computers and Digital Techniques*, vol. 153, no. 3, pp. 157 – 164, 2006.

[2] K. Papadimitriou, A. Dollas, and S. Hauck, "Performance of partial reconfiguration in FPGA systems: A survey and a cost model," *ACM TRETS*, vol. 4, no. 4, pp. 36:1–36:24, Dec. 2011.

[3] K. Compton and S. Hauck, "Reconfigurable computing: a survey of systems and software," *ACM Computing Surveys (csuR)*, vol. 34, no. 2, pp. 171–210, 2002.

[4] V. Betz, J. Rose, and A. Marquardt, Eds., *Architecture and CAD for Deep-Submicron FPGAs*. Norwell, MA, USA: Kluwer Academic Publishers, 1999.

[5] L. McMurchie and C. Ebeling, "Pathfinder: A negotiation-based performance-driven router for FPGAs," in *FPGA*, 1995, pp. 111–117.

[6] I. Sourdis, J. Bispo, J. Cardoso, and S. Vassiliadis, "Regular expression matching in reconfigurable hardware," *Journal of Signal Processing Systems*, vol. 51, pp. 99–121, 2008.

[7] Bleeding edge threats website. [Online]. Available: http://www.bleedingthreats.net

[8] S. Yang, *Logic synthesis and optimization benchmarks user guide: version 3.0*. Citeseer, 1991.

[9] G. Lemieux, E. Lee, M. Tom, and A. Yu, "Directional and single-driver wires in fpga interconnect," in *Field-Programmable Technology, 2004. Proceedings. 2004 IEEE International Conference on*. IEEE, 2004, pp. 41–48.

[10] J. Rose, J. Luu, C. W. Yu, O. Densmore, J. Goeders, A. Somerville, K. B. Kent, P. Jamieson, and J. Anderson, "The VTR project: architecture and CAD for FPGAs from verilog to routing," in *Proceedings of FPGA*. ACM, 2012, pp. 77–86.

[11] Altera, *Engineering Change Management with the Chip Planner*, 2012.